SECURITY FOR DEVOPS DEPLOYMENT PROCESSES: DEFENSES, RISKS, RESEARCH DIRECTIONS

Norman Wilde, Brian Eddy, Khyati Patel, Nathan Cooper, Valeria Gamboa, Bhavyansh Mishra, Keenal Shah

Department of Computer Science, University of West Florida, Pensacola, Florida, USA

ABSTRACT

DevOps is an emerging collection of software management practices intended to shorten time to market for new software features and to reduce the risk of costly deployment errors. In this paper we examine the security implications of two of the key DevOps practices, automation of the deployment pipeline using a deployment toolchain and infrastructure-as-code to specify the environment of the deployed software. We focus on identifying what changes when an organization moves from manual deployments to DevOps automated deployment processes.

We reviewed the literature and conducted three case studies using simple configurations of common DevOps tools. This allowed us to identify specific:

- *Positive influences on security where automation enhances defenses.*
- *Negative influences, where automation enables different kinds of attacks and increases the attack surface.*
- *Research directions that look promising to support this new approach to software management.*
- *Recommendations for DevOps adopters*

KEYWORDS

DevOps, cloud computing, security, infrastructure-as-code, continuous delivery, continuous deployment, continuous security, release engineering, SecDevOps, DevSecOps

1. INTRODUCTION

In the last few years a new approach called DevOps has emerged for the management of large cloud-hosted software applications. The portmanteau word DevOps signifies the integration of the activities of software development with those of operations. Broadly speaking, the intent is to reduce time to market for new software features and to eliminate potentially costly deployment errors.

Different authors present different definitions of DevOps [1]. It is generally seen as a collection of mutually-reinforcing practices that responds to the differing motivations of development and operation teams. Developers seek rapid change and the opportunity to put new software features in place as quickly as possible while operations staff wants stability to control risk [2]. DevOps practices range from the cultural (e.g. treat Ops as first-class citizens) to the technical (e.g. continuous deployment) [3].

In this paper we will focus on the security implications of two key DevOps practices: automation of the deployment pipeline and infrastructure-as-code to specify the environment of the deployed software.

DOI : 10.5121/ijsea.2016.7601

A DevOps continuous deployment pipeline is an automated toolchain. When code is checked in by a developer, a series of steps take place with little or no manual intervention. The new software is built by one set of tools and deployed to a test environment which is provisioned by another set of tools. Other tools run the tests and, if these pass, the new code may then be released to staging and production environments [4].

DevOps is enabled by cloud computing and, indeed, would be almost inconceivable without infrastructure-as-a-service clouds. DevOps requires multiple near-identical execution environments, so that developers, testers, security analysts and so on see essentially the same environment as the end-user facing production.

It is the infrastructure-as-code concept that keeps these environments consistent. Each environment, including its virtual hardware, its virtual networks, and its system software, is specified by templates and scripts that make up the infrastructure-as-code. The templates and scripts are baselined and version controlled just as is done with application code [5]. To the greatest extent possible, the same templates and scripts are used across the different environments, thus guaranteeing consistency. Executing the scripts creates copies of the environment that are provisioned, used, moved to production or discarded, often hundreds of times a day. This flexibility is strongly supported by cloud computing, with its ability to create virtual hardware rapidly and cheaply.

This paper is an attempt to review systematically the security issues associated with moving to a DevOps process. Thus, we do not look at cloud security issues in general. Rather, we focus on the question of what changes when one moves from manual cloud deployments to DevOps automated cloud deployments.

Of necessity, the paper involves an element of speculation in this rapidly emerging field. This work describes our understanding of current DevOps practices. It is based on review of available tool documentation, attendance at industry courses, conversations with professionals in the field and on our own case studies of three different toolchains (Section 4). However our results cannot be comprehensive since there are many other tools in use, DevOps practices are diverse, and security challenges are constantly changing.

The paper is organized as follows. In the next section, we describe DevOps and the infrastructure-as-code concept in greater detail. Then in Section 3 we go on to review related work on DevOps and security.

In Section 4 we describe our three case studies. In each study we created a toolchain using a different set of widely-used DevOps tools. We then prepared and ran infrastructure-as-code scripts and templates that use the toolchain to perform a simple software deployment. From an analysis of the three cases we identify commonalities and construct an abstract toolchain model. We then define a risk model of the potential threat and use it to discuss the available security controls for each toolchain.

From this experience, in Section 5 we summarize the ways DevOps affects security and identify:

- Positive influences on security where automation enhances defenses and thus would appear to facilitate good overall application security.
- Negative influences, where automation enables different kinds of attacks and increases the attack surface that needs to be protected.
- Research directions that look promising to support this new approach to software management.
- Specific recommendations for DevOps adopters.

Finally, we summarize our conclusions in Section 6.

2. DEVOPS BACKGROUND

A key goal of DevOps is to greatly reduce the risk of expensive configuration errors in the production environment stemming from human errors during deployment. A commonly cited case is that of Knight Capital, an American high frequency securities trading company [6]. On August 1, 2012 Knight installed updated software on its systems, but due to a human error only 7 of the 8 servers received the new version. When trading started, the old version misinterpreted a flag which had been repurposed in the new version, leading that one server to go into an infinite loop rapidly sending out buy and sell orders. After about 45 minutes and 4 million unwanted transactions Knight had lost $460 million and was forced out of business.

DevOps attempts to reduce the risk of such errors using an automated continuous deployment pipeline, shown schematically in Figure 1. The concept is to have a single set of infrastructure-as-code (scripts, templates, etc.) that runs on one or more build servers to create all the organization's software environments, from development through several testing environments to staging and then to production. The continuous deployment pipeline may be initiated either manually by a developer or automatically after a code change by an automation server such as Jenkins [7].

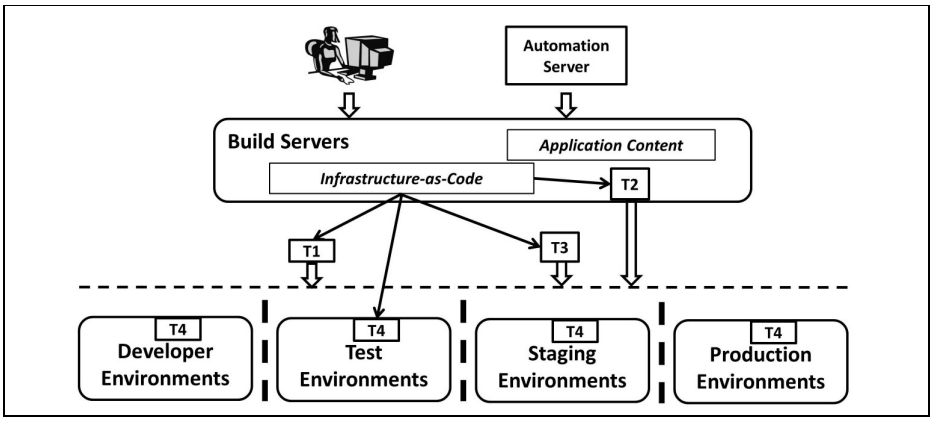

Figure 1. Schematic View of Tools in an Automated Continuous Deployment Pipeline

New application code moves through the pipeline and passes through several "gates" where it is tested or verified automatically. At each point the infrastructure-as-code may provision an environment of new cloud virtual machines, deploy the application to them, run tests or scans, and destroy the machines when they are no longer needed. Though there are some inevitable differences, the infrastructure-as-code keeps all environments as similar as possible.

Each environment is created by a collection of tools, here called the toolchain. Some of these tools run on the build server (e.g. T2 in the diagram), some are external services (T1, T3), while others run on the provisioned virtual machines (T4). The tools for provisioning and application deployment are kept the same for all environments.

An emerging practice is to utilize immutable virtual machines, sometimes called "fully baked". These virtual machines are created with one or more applications installed on them, and then left unchanged. If any update is needed the old virtual machine is discarded and a new one is created using the deployment pipeline. System administrators never log in to production machines. We will point out later in Section 5 that this practice has significant security benefits.

3. RELATED WORK

There is very little published academic work on security as it relates to DevOps. In an early paper (2011) Merkow and Raghavan discussed practices for what they call "continuous security" across the System Development Life Cycle [8]. Their paper was written before DevOps concepts became common, but many of their suggestions are relevant, particularly as to the use of security tools and their incorporation into the build process. Fitzgerald and Stol in a survey paper on continuous software engineering highlighted a number of activities that must become "continuous" in the DevOps approach [9]. One of these activities is "continuous security" which must become a key concern throughout the development process. As an example, they mentioned an experiment with a modified Scrum process in which regulatory compliance was assessed at the end of each Scrum sprint, instead of just at the end of the development cycle. Weber, Nepal and Zhu mentioned the need to secure the continuous deployment pipeline and the desirability of incorporating static and runtime security checks, but they do not go into details [10].

In their book on enterprise software security, van Wyk, Graff, Peters and Burley focused on the organizational view of DevOps and discussed how IT Security personnel should work with developers, for example in handling different kinds of security events [11]. They referred to this approach as "rugged DevOps". They mentioned, for instance, that during the handling of a security event the DevOps toolchain may need to be used to set up a special security incident test bed whose firewalls and intrusion detection must closely match the production settings so that an attack can be replayed or simulated for analysis.

Bass, Weber and Zhu devoted a chapter of their book on DevOps to security and security audits [3]. Most of their discussion would be applicable equally to traditional and DevOps deployments, but the authors pointed out a number of specific concerns, such as the need to protect the deployment pipeline, the need to make sure old virtual machines are removed, the benefits of including automated security testing in the deployment pipeline, and the desirability of using "fully baked" servers with a short life span.

The papers by Rimba et. al. and Bass et. al. used DevOps toolchains as examples in discussing a particular approach to securing software. Rimba et. al. focused on using design patterns for security and included a case study involving a DevOps toolchain [12]. Similarly, Bass et. al. took a DevOps toolchain as their example in describing their process for hardening a software system by identifying trusted and untrusted components, and then introducing trusted components that mediate access to the untrusted components [13]. Neither paper sought to analyze security issues associated with DevOps in general. These papers may be taken, however, as evidence that security in DevOps is a growing concern.

Industry concern with DevOps is further shown at industry-focused conferences where courses on DevOps and its security are becoming common. The terms SecDevOps or DevSecOps are now being used at these conferences. For example, at the Software Engineering Institute's SATURN 2015 conference Bellomo, Cois, and Kazman gave a full day course on DevOps with a major component on security. The presentation focused on security "anti-patterns" and how to avoid them. One theme (which we have already mentioned) was the desirability of using immutable or "fully baked" virtual machines that are never administered after deployment, so vulnerabilities may not be introduced, accidentally or deliberately, without going through the approved build process. Another theme was the need for automated security testing during the build process instead of relying on manual penetration testing of the deployed system.

A one day DevOps course at the 2015 Amazon AWS™ ReInvent conference was titled "Securing Next-Generation Workloads at Cloud Scale", and went into considerably greater detail, but focused specifically on the different AWS services and how they could contribute to security. One interesting concept was that role conflicts between developer personnel and IT security personnel could be mitigated by having each side "own" some part of the infrastructure-as-code. As an example, the template used in provisioning virtual machines and their environment could be divided into two parts: developers specify the virtual machines and the application deployment steps; IT security personnel specify the firewall settings. At deploy time, a master script weaves together the two parts of the template before the AWS provisioning service is called.

We will return to some of these points in our discussion in Section 5, but first we will describe our three case studies.

4. CASE STUDIES

4.1. Toolchain Selection and Configuration

The purpose of the case studies was to identify specific commonalities and differences in a range of DevOps toolchains with respect to attack surface and security controls. Real DevOps toolchains can be very complex [4] but to keep the analysis feasible those used in our case studies were limited in scope. First, the use case for all three studies was a very simple scenario of provisioning an immutable virtual machine and deploying on it an application consisting of a simple "hello world" web site served by the Python Simple HTTP Server. We did not study other possible toolchain functions such as triggering deployment after a code change, distributing application updates, running automated tests, or scaling resources based on load. For consistency, in all three cases the deployment environment was Amazon EC2™, the current market leader. Finally, since many of the DevOps tools can be used in a variety of ways and combinations, we had to pick one possible toolchain configuration from the many possibilities. In general, we tried to keep our configuration as simple as possible while following suggested good practices from published tutorials and from the tool vendor.

We defined three toolchains, combining tools that are frequently mentioned in DevOps discussions. However to expose the widest range of security issues we picked tools with different architectures and different approaches to application deployment:

- **Toolchain A - CloudFormation, CodeDeploy, S3, Bash**: This toolchain calls commercial API's for provisioning and deployment. AWS CloudFormation™ is Amazon's service for provisioning virtual machines (called "instances") and other virtual hardware [14]. Amazon CodeDeploy™ is one of their services to automate code deployments to an instance [15]. Amazon S3™ is a general cloud storage service and Bash is a popular shell command language for Unix machines. All of the AWS services may be easily controlled by Bash scripts that call the AWS Command Line Interface (CLI).

- **Toolchain B - Chef Server, Chef Knife, Chef Client, Knife EC2 Plugin, Bash**: Chef is the name of a suite of open source IT automation tools supported by Chef Software Inc. [16]. Greatly simplified, to use Chef a developer writes "recipes" that describe the desired state of some resource on a virtual machine. Chef Server manages the recipes and makes them available. We used a Chef Server EC2 instance launched from the standard Chef Amazon Machine Image (AMI). Chef Knife and Chef Client work with the recipes

running on the developer side or the virtual machine side respectively. The Knife EC2 Plugin adds functionality to Knife to work with AWS EC2 instances.

- **Toolchain C - Docker, CloudFormation, S3, Bash**: Docker differs from the tools used in the first two case studies in that it deploys applications to lightweight containers instead of to complete virtual machines. Each container is isolated from others running on the same machine, but the container holds everything needed by an application including code, data, libraries, etc. Claimed advantages of containers include faster start-up and better sharing of machine resources [17].

4.2. Toolchain Model

When the studies were completed and documented, we compared the three toolchains to model commonalities. The following abstract components were identified (Figure 2):

- **Build Server** - the computer where the infrastructure-as-code is executed to provision the virtual machine and to build and deploy the application. The build server draws from a repository to obtain the:
    - **Infrastructure-as-Code** - the scripts and templates that describe the virtual machine, its network environment, and the process for building and deploying the application.
    - **Application Content** - the files making up the application, including source code, web pages, data, etc.
- **Provisioning Service** - the service that actually instantiates the Provisioned Virtual Machine and its environment.
- **Tool Store** - holds code for tools that need to be downloaded to the Provisioned Virtual Machine, mainly the Deployment Agent.
- **Artifact Store** - holds the build artifacts that need to be downloaded to the provisioned virtual machine.
- **Deployment Service** - the service that knows which applications go on which virtual machines and mediates between the virtual machine and the artifact store.

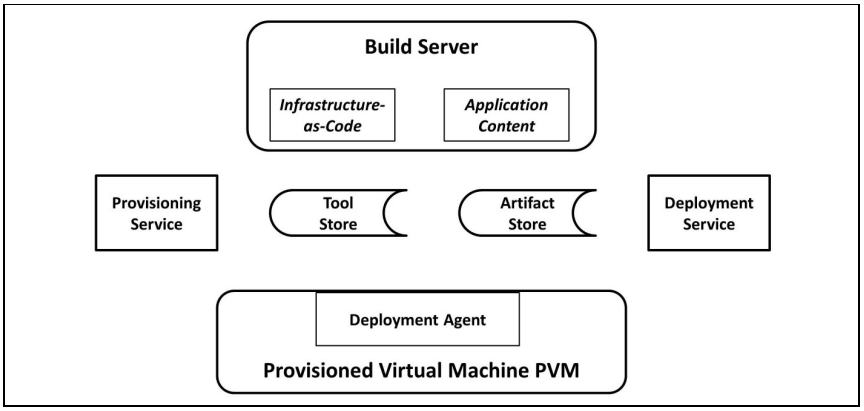

Figure 2. High Level Model of Toolchain Components

- **Provisioned Virtual Machine (PVM)** - the virtual computer created to run the application. This virtual machine also hosts the:
    - **Deployment Agent** - software that is "bootstrapped" onto the provisioned virtual machine and which then takes over local tasks (copying files, starting services, etc.) needed to start the application.

With a few exceptions, each of the toolchains contains these components, but the names used by tool vendors vary widely. For example Chef calls the Build Server the "workstation", the Provisioned Virtual Machine is a "node" while the Deployment Agent is a "client".

4.3. Security Risk Model

For a security analysis of the three toolchains, we consider a risk model as shown in Figure 3 and follow the risk analysis terminology of [18]. We hypothesize a sophisticated adversarial threat source of high capability that could be a group or a nation state with significant resources to devote to an attack.

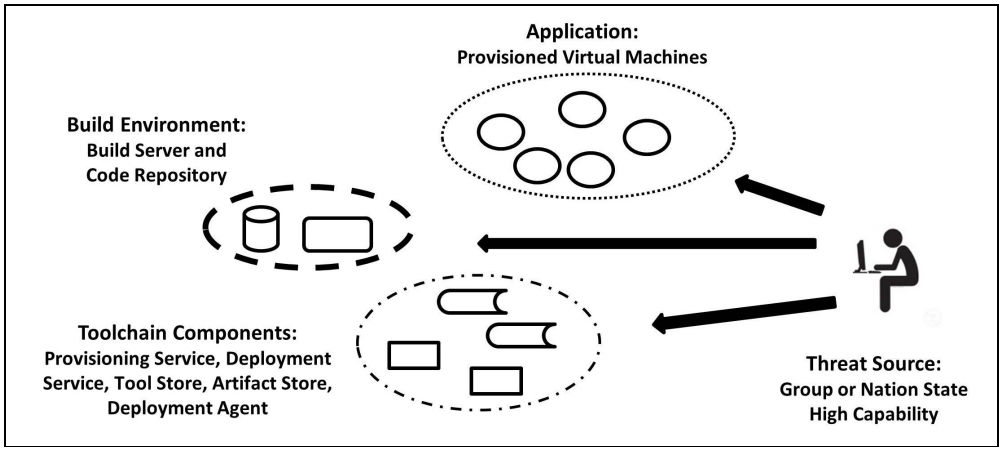

Figure 3. Security Risk Model

Our hypothetical organization maintains an Application that includes the PVMs deployed by the continuous deployment pipeline. As indicated in the figure, the perimeter defense of the Application is relatively weak since it is accessible over the public internet. The Build Environment contains the organization's code repository and the Build Server(s) that run the pipeline. The Build Environment's perimeter is more strongly defended because access to these components can be restricted to the organization's personnel and its private network.

Both the application and the Build Environment would exist even in a manual deployment process. What is new in a DevOps process are the toolchain components: the Provisioning Service, the Deployment Service, the Tool Store, the Artifact Store, and the Deployment Agent. These constitute additional attack surface that is exposed to the threat source. A successful attack on any one of these new components could be very costly, since a compromised component could let the attacker read and possibly modify the software for many of the organization's applications. A sophisticated attack on these components could be designed to ignore development and test environments and to trigger only when working with staging or production PVMs. At that point backdoor malware could be added undetected.

The three toolchains use very different procedures for provisioning and deployment. The infrastructure-as-code is different, the sequence of events is different, and the security controls employed are different. We will examine each of our implementations of the toolchains in greater detail. In each case we can classify the observed security controls as authentication/authorization controls, firewall controls, and transport layer security controls.

4.4. Toolchain A - Security Analysis

The infrastructure-as-code consists of a CloudFormation infrastructure template and a top level Bash deployment script. The template contains an embedded user-data Bash script which runs on the provisioned machine at start-up. Additionally, the application content contains the web page, an AppSpec file with deployment instructions, and a start-up script to run the web server. The deployment script initiates the following sequence of events (Figure 4):

1) The deployment script zips the application content and uploads it to a user bucket in S3.
2) The deployment script calls the CloudFormation service and passes it the template.
3) CloudFormation provisions the PVM.
4) As the PVM boots it runs the user-data script to download the CodeDeploy Agent from an Amazon bucket in S3 and install it.
5) The deployment script calls the CodeDeploy service to create a "deployment" with information about the application and its location in S3.
6) The CodeDeploy Agent queries the CodeDeploy service to locate the application content.
7) The CodeDeploy Agent retrieves the application content from S3, downloads it, and extracts the AppSpec file.
8) The CodeDeploy Agent follows the instructions in the AppSpec file to deploy the web page and start the HTTP server.

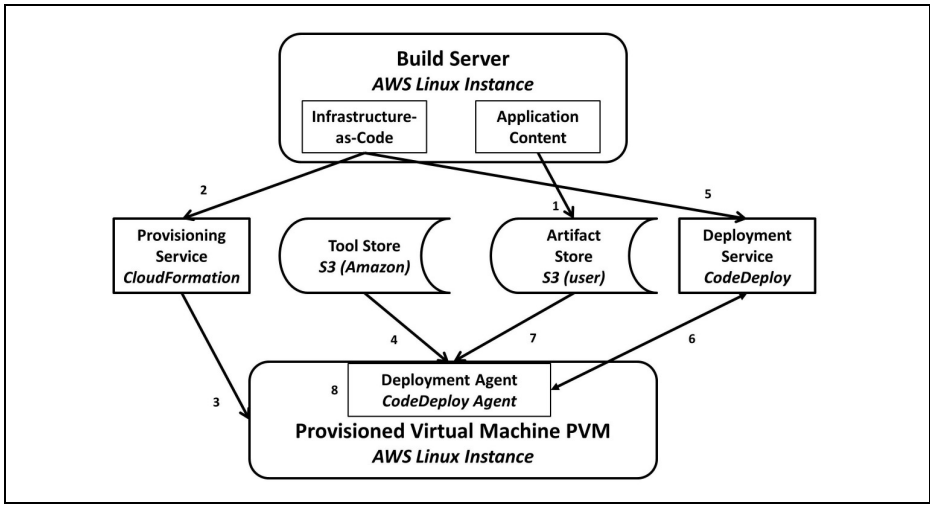

Figure 4. Toolchain A - Sequence of Events

Toolchain A, being largely based on Amazon API's, uses Amazon's Identity and Access Management (IAM) service for its authentication/authorization controls [19]. IAM allows the definition of users and roles. Both of these are assigned policies that specify how they can access AWS resources. For the Toolchain A study, an IAM user is issued credentials to run the scripts in the infrastructure-as-code while IAM roles are used for service-to-service authentication/authorization. An instance profile role is attached to the PVM and allows the

CodeDeploy Agent to read application content from S3. A service role is passed to the CodeDeploy Service to allow it to scan the organization's EC2 instances and identify which of these need software deployments.

IAM policies allow sophisticated, fine-grained security control, but they suffer from complexity and limited transparency. The policies are stored in multiple places within IAM so they are not visible to a developer inspecting the infrastructure-as-code. The settings are also complex; getting them all right was a major challenge in doing the case study. A naive developer will be tempted to grant very broad access to get an application to run.

There are several possible new attacks that could be mentioned. If the IAM user's credentials were compromised the attacker could deploy malicious code through CloudFoundation, CodeDeploy or S3. A compromised PVM could read from the S3 user bucket and may thus be able to steal copies of the organization's software. And of course, any deep compromise to the security of any of CloudFoundation, CodeDeploy or S3 could have broad impact on the organization.

AWS allows a firewall (called a security group) to be attached to each virtual machine. For Toolchain A that allows some protection of the PVM, though the application that is deployed to it will require some ports to be open. AWS uses HTTPS for transport layer security of all communications.

4.5. Toolchain B - Security Analysis

The Build Server has installed on it the Chef Development Kit including the Knife command line tool and a Knife EC2-Plugin. The infrastructure-as-code consists of a Knife configuration file and a deployment script. The application content consists of a Chef recipe that describes the web site and specifies how to start the web server.

The sequence is as follows (Figure 5):

1) The deployment script calls Knife to upload the recipe to the Chef Server, which will serve as both Artifact Store and Deployment Service.
2) The deployment script calls the Knife EC2 Plugin
3) The EC2 Plugin provisions the PVM.
4) The EC2 Plugin then logs in to the PVM.
5) The EC2 Plugin instructs the PVM to download and install the Chef Client from a Chef package repository which acts as the Tool Store.
6) The Chef Client then queries the Chef Server to get the recipe for the application.
7) The Chef Client runs the recipe to deploy the web site and to start the web server.

For Toolchain B as with Toolchain A, an IAM user is issued credentials to run the scripts in the infrastructure-as-code. However, thereafter Chef relies more on public-private key pairs for authentication/authorization. Chef Server issues two pairs of keys to the Build Server as part of the configuration of the Chef Development Kit. These key pairs are used when the Build Server authenticates to the Chef Server. Additionally, as each PVM makes its first Chef run, the Chef Server issues it a unique key pair for use in future authentication/authorization [20].

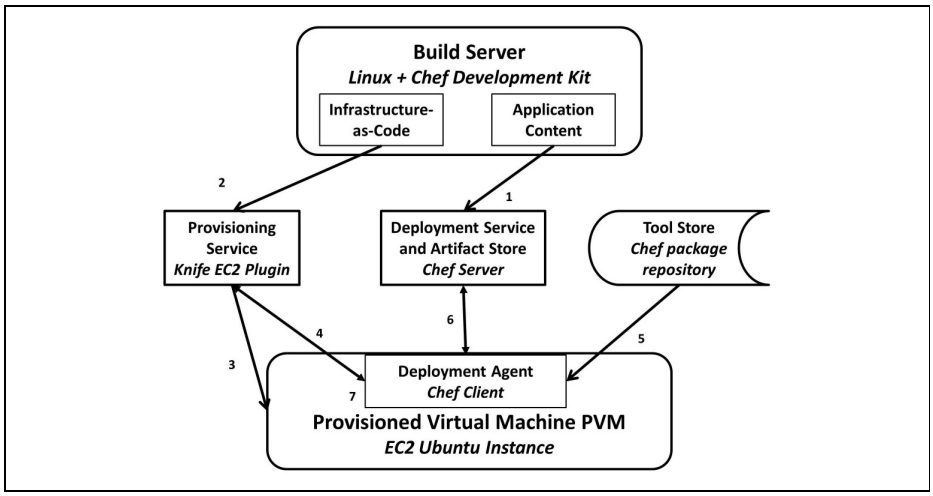

Figure 5. Toolchain B - Sequence of Events

We found this mechanism to be easier to set up than the IAM roles of Toolchain A and, since the key pairs are created automatically, there may be less risk that over-broad policies are assigned. However, a main difficulty may be that any future change to use a different Chef Server instance could be very complicated as the old key pairs would no longer be valid.

With Toolchain B the interactions between the Build Server, Chef Server and the PVM are more complicated than with Toolchain A. With Chef, Knife on the Build Server uploads recipes to the Chef Server, and the Knife EC2 Plugin then logs in to the PVM to bootstrap the Chef Client. Then, the PVM has to communicate with the Chef Server to get the recipes. While creating complications, this three way communication does provide an opportunity to lock things down using AWS firewalls (security groups). Unfortunately the firewall settings recommended for use with the Chef Server standard AMI are extremely broad. After some study, we were able to replace these with a fairly complicated three way set of firewall settings to provide least privilege access.

As with Toolchain A, an attacker who succeeds in compromising the PVM could go on to penetrate the deployment pipeline. The key pair and firewall settings would not prevent the opponent from reaching the Chef Server, so that an initial lodgement on the PVM could provide a base for attack.

Also as with Toolchain A, Chef uses SSH or HTTPS for transport layer security on all communications. However, the Chef Server launched from the standard Chef AMI issues itself a self-signed certificate, which opens opportunities for a man-in-the-middle attack.

4.6. Toolchain C - Security Analysis

The infrastructure-as-code consists of a CloudFormation infrastructure template and a Bash deployment script which runs on the Build Server, a Dockerfile, which describes the Docker container, and a Bash helper script. The helper script installs Docker on the PVM and deploys the application into the Docker container. The application content is a simple web page.

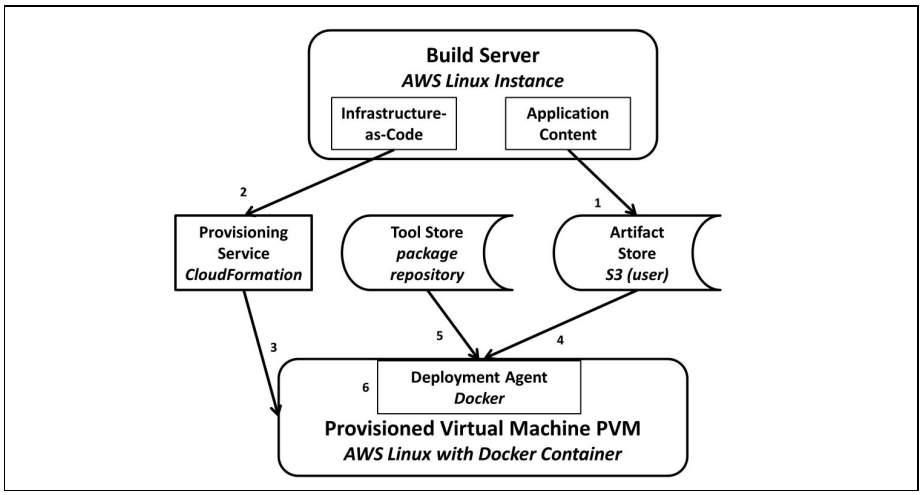

Figure 6, Toolchain C - Sequence of Events

The sequence is controlled by the deployment script and is as follows (Figure 6):

1) The deployment script uploads the application content, the Dockerfile, and the helper script to a bucket in S3 which acts as the Artifact Store.
2) The deployment script calls the CloudFormation provisioning service, passing it the template.
3) CloudFormation creates the PVM, which is an Amazon Linux instance.
4) As the PVM starts up, a user data script embedded in the template runs, downloads the application content, the Dockerfile and the helper script from S3, and then runs the helper script.
5) The Bash helper script installs Docker from a package repository to the PVM and runs it. Docker creates a Docker image based on the Dockerfile passed to it and deploys a Docker container made using the newly created Docker image.
6) A python HTTP server running inside the Docker container serves the web page on port 8080 of the Docker container, which has been mapped to port 80 of the PVM

This toolchain also uses Amazon IAM for authentication. An IAM user is needed to run the deployment script; this user needs permissions to write to S3, to run CloudFormation, and to pass an instance profile role to the PVM. The instance profile role lets the PVM read the application content from S3. One security group was needed, to open the PVM's port 80 so it could serve the web page. HTTPS is used in communicating from the Build Server to the CloudFormation and S3 services.

In some ways this toolchain was rather simpler in its security configuration than the preceding examples. It took some study to get least-privilege permissions for the IAM user and role right, but the firewall settings were quite straightforward. No Deployment Service was needed which reduced complications and the security risks associated with PVM communications back to the toolchain components.

5. DEVOPS SECURITY DISCUSSION

5.1. DevOps New Defenses

While there are problems, DevOps processes would seem to have the potential to significantly improve security defenses. If application structure and environment are represented in infrastructure-as-code, then theoretically that code can be inspected, tested and managed just as is source code. (As we will note in subsection 5.3 the current format of the code can make security analysis difficult.) There is a reduced chance of security vulnerabilities introduced through simple manual error. If automated security tests are available, these can be performed concurrently in duplicated test environments that are guaranteed to be very similar to the production environment. The DevOps advantages are substantially enhanced if the organization adopts the practice of using short-lived immutable virtual machines. With this practice the whole Application may be recreated every 24 hours or so. There is no manual security patching of production machines; instead each new build can start from an up-to-date image housing the latest patched operating system and middleware. Virtual machines are "fully baked" and never maintained so a malicious or careless employee cannot plant malware in the production environment without going through the build approval process. Perhaps most important, frequent replacement of virtual machines makes life much harder for sophisticated attackers who first break into the Application and then establish long term persistence by planting backdoor malware [21]. The backdoor would be eliminated each time the virtual machine was replaced.

5.2. DevOps New Risks

As we compare DevOps automated cloud deployment with manual cloud deployment it is clear that there are some new risks that need to be managed. We have already mentioned the additional attack surface that is provided by the Provisioning Service, the Tool Store, the Artifact Store, and the Deployment Service. Any weak link in the toolchain is a potential entry for a very damaging attack. A specific new risk that we observed in Toolchains A and B is that the PVMs that make up the Application may have a channel back to these new components. According to our risk model, the Application has a relatively lightly defended perimeter so that there is some risk that a compromised PVM can serve as a starting point for a dangerous attack.

Some of the attack surface risk can be mitigated by careful attention to available firewall and authentication/authorization mechanisms. However in doing the case studies we noted that the tools are complex; a single toolchain often has multiple interdependent code files in several different file formats. Current documentation is often silent about "least privilege" security settings; in fact tutorials often suggest assigning very broad permissions. It can thus be quite difficult to establish exactly the minimum policies required so that a tool can run correctly. So as DevOps becomes widespread there is a danger that many organizations may adopt these tools without learning to adequately lock down their systems.

With the frequent automated deployments of DevOps, traditional manual approaches to security, such as hand-evaluated security assurance cases [22], become much less practical. Given the time constraints, automated methods of evaluating security properties must predominate. But it is not possible to automate all security checks, and the reduced number of human eyes on the process certainly must raise concerns.

Key management is a problem, even with manual system administration, but it may be even more difficult with DevOps automation. The PVMs need keys and certificates for database access, for transport layer security, and so on. Traditionally such keys were provided by a system

administrator logging in and configuring the keys by hand. There may be many keys required and it is not easy to automate this kind of key management safely. A commonly reported error is to place a key within a script which is then checked into a source code control system and thus becomes visible to all members of the development team.

Several of the toolchains use public components from their "community". For example, in all of our three toolchains the Deployment Agent was downloaded from a public site. Additionally, some of the toolchains have public sets of code, templates or scripts, some of which seem to get installed on the Build Server or the PVM with little warning. It may be hard for an organization to know exactly what code they are using and where it came from.

5.3. Research Directions

One of the main claimed benefits of DevOps infrastructure-as-code is that the code can be inspected, tested and managed. From a security perspective however, this is more an aspiration than a reality. We have already mentioned that, even within one toolchain, the code is complex and is split into multiple files with interdependencies and different formats. In our three case studies the configuration of the security controls was even further dispersed and resided partially in Amazon's IAM service.

Security configuration is not visible in the templates and scripts themselves. For example in the Toolchain A case study, the policies and trust relationships needed a total of 9 JSON fragments, some written by us and some provided by AWS, but all accessible only via IAM. Suppose an auditor of our code should ask "what S3 buckets can the PVM access?" To answer that question she would have to trace through the master deployment script, identify which IAM role is assigned to the PVM, go to Amazon's AWS console and find that role, and then scan each of the policies attached to the role.

A promising research goal would seem to be to structure the infrastructure-as-code so that security questions can be more easily answered. Simple navigation, search and dependency analysis tools could be a good place to start. A clearer separation of concerns would help, so that security experts could examine the code for security issues while application experts focus on the mechanics of provisioning and deployment. A still greater advance would be some compact way of representing infrastructure-as-code that would allow proofs of security properties. In other words, infrastructure-as-code could benefit from good programming languages with appropriate specification and assertion mechanisms.

Another research goal would be to enrich the set of feasible automated security checks. DevOps build chains often run audit and validation tools before each new release is promoted to the production environment. Anything that could improve the scope and precision of these tools would be very useful. Performance is also important. Since there may be many releases each day, tools should be designed to work incrementally so that each release does not require a full system scan.

Better intrusion detection could also help alleviate the risks of attacks on DevOps components. For example the Build Server is a dangerous target, but it should be running a very limited and specific set of services. Research could identify which intrusion detection strategies are best for this and other DevOps components.

5.4. Specific Recommendations

While every organization will face different circumstances, our review of related work and our case studies would seem to indicate that adopters of DevOps automation should consider at least the following policies:

1) Where application structure makes it possible, use immutable PVMs and replace them often.
2) If possible, block the path from the PVM back to tools such as the Provisioning Service, the Artifact Store, and the Deployment Service. This will make it harder for a compromised PVM to be used as a lodgement for attacks on the toolchain.
3) The Build Server and the other toolchain components are likely to become more prominent targets. The activity on these components should be logged and the logs closely monitored to detect an attack as early as possible.

6. CONCLUSIONS

In this paper, we have tried to identify some of the security benefits and drawbacks of the DevOps approach to managing software. We have focused on how infrastructure-as-code allows automated provisioning of infrastructure and automated deployment of applications. We performed three case studies using DevOps tools that are in widespread use and identified the changes automation brings to the organization's attack surface. We also examined the security controls provided by the DevOps tools used in our studies.

Our study was necessarily limited since there are many other tools used in DevOps; even with the tools we picked we had to choose one specific way of using each tool and that often excluded some of its features. Our results will not necessarily generalize to all DevOps tools and implementations.

However, we think one conclusion is generally valid: that current DevOps tooling suffers from complexity. Using the tools, even those from a single vendor, involves using interdependent templates and scripts in multiple formats. The code invokes numerous functions and APIs that are not clearly documented. Security policies are not visible in the infrastructure-as-code but are stored externally in formats that require an expert for interpretation. Thus a developer attempting to build a secure system must have deep guru-level knowledge in multiple areas.

Thus, in this paper we do not argue that DevOps makes cloud computing less secure. On the contrary there are some significant security benefits and the potential attack points we have described might all be managed by well qualified personnel acting with care and attention. Rather we argue that if, as seems likely, DevOps practices spread widely, it will be difficult to have available enough personnel with the required skills and focus. So we believe that a main direction for research should be to simplify security aspects of DevOps so that they can be managed by developers of normal skill.

ACKNOWLEDGEMENTS

Work described in this paper was partially supported by the University of West Florida Foundation under the Nystul Eminent Scholar Endowment. "Amazon AWS", "EC2", "CloudFormation", "CodeDeploy" and "S3" are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.

REFERENCES

[1] Andrej Dyck, Ralf Penners, and Horst Lichter, "Towards definitions for release engineering and DevOps," in Proceedings of the Third International Workshop on Release Engineering (RELENG '15), 2015, IEEE Press, Piscataway, NJ.

[2] Michael Hüttermann, DevOps for Developers, 2012, Apress, ISBN 978-1-4302-4569-8 (Print) 978-1-4302-4570-4 (Online).

[3] Len Bass, Ingo Weber, and Liming Zhu, Devops: A Software Architect's Perspective (1st ed.), 2015, Addison-Wesley Professional, ISBN:0134049845.

[4] Derek E. Weeks, "31 Reference Architectures for DevOps and Continuous Delivery," http://devops.com/2015/04/22/31-reference-architectures-devops-continuous-delivery/, {Link accessed Oct. 20, 2016}.

[5] Sanjeev Sharma, "Understanding DevOps – Part 5: Infrastructure as Code," https://sdarchitect.wordpress.com/2012/12/13/infrastructure-as-code/, {Link accessed Nov. 2, 2016}.

[6] Securities and Exchange Commission, "In the Matter of Knight Capital Americas LLC," https://www.sec.gov/litigation/admin/2013/34-70694.pdf {Link accessed Oct. 20, 2016}.

[7] Jenkins, "Jenkins Documentation," https://jenkins.io/doc/ {Link accessed Oct. 20, 2016}.

[8] Mark Merkow and Lakshmikanth Raghavan, "An Ecosystem for Continuously Secure Application Software," CrossTalk, March/April 2011, pp. 26 - 29, http://static1.1.sqspcdn.com/static/f/702523/10957338/1298784313433/201103-Merkow.pdf, {Link accessed Oct. 20, 2016}.

[9] Brian Fitzgerald and Klaas-Jan Stol, "Continuous software engineering and beyond: trends and challenges," in Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE 2014). ACM, New York, NY, USA, 1-9. DOI= http://dx.doi.org/10.1145/2593812.2593813.

[10] I. Weber, S. Nepal and L. Zhu, "Developing Dependable and Secure Cloud Applications," IEEE Internet Computing, vol. 20, no. 3, pp. 74-79, May-June 2016, doi: 10.1109/MIC.2016.67.

[11] Kenneth R. van Wyk, Mark G. Graff, Dan S. Peters, and Diana L. Burley, Enterprise Software Security: A Confluence of Disciplines (1st ed.), 2014, Addison-Wesley Professional, ISBN-13: 978-0321604118.

[12] P. Rimba, Liming Zhu, L. Bass, I. Kuz and S. Reeves, "Composing Patterns to Construct Secure Systems," Eleventh European Dependable Computing Conference (EDCC), Paris, 2015, pp. 213-224. doi: 10.1109/EDCC.2015.12.

[13] Len Bass, Ralph Holz, Paul Rimba, An Binh Tran, and Liming Zhu. "Securing a deployment pipeline," in Proceedings of the Third International Workshop on Release Engineering (RELENG '15), 2015 IEEE Press, Piscataway, NJ, USA, 4-7.

[14] Amazon Web Services, "AWS CloudFormation," https://aws.amazon.com/cloudformation/ {Link accessed Oct. 20, 2016}.

[15] Amazon Web Services, "AWS CodeDeploy," https://aws.amazon.com/codedeploy/ {Link accessed Oct. 20, 2016}.

[16] Chef Software Inc., "About Chef," https://www.chef.io/about/ {Link accessed Oct. 20, 2016}.

[17] Docker, "What is Docker," https://www.docker.com/what-docker, {Link accessed Oct. 20, 2016}.

[18] National Institute of Standards and Technology, Guide for Conducting Risk Assessments, NIST Special Publication 800-30, Revision 1, http://dx.doi.org/10.6028/NIST.SP.800-30r1.

[19] Amazon Web Services, "AWS Identity and Access Management (IAM)," https://aws.amazon.com/iam/, {Link accessed Oct. 20, 2016}.

[20] Chef Software Inc., "Authentication, Authorization," https://docs.chef.io/auth.html, {Link accessed Oct. 20, 2016}.

[21] Rob Joyce, "NSA TAO Chief on Disrupting Nation State Hackers," USENIX Enigma 2016, http://youtu.be/bDJb8WOJYdA, {Link accessed Oct. 20, 2016}.

[22] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw and N. R. Mead, Software Security Engineering: A Guide for Project Managers, Addison-Wesley, 2008, ISBN-13: 978-0321509178.

Authors

Norman Wilde is a Professor and Nystul Chair in the Computer Science Department of the University of West Florida. He received his PhD from the Massachusetts Institute of Technology in Mathematics in 1971. His research interests are Software Engineering, Software Evolution, Services Oriented Architectures, and Cybersecurity.

Brian Eddy is an Assistant Professor in the Department of Computer Science at the University of West Florida. He graduated with his PhD from The University of Alabama in 2015. His primary research interests include Software Maintenance and Evolution, Program Comprehension, and Agile Software Development.

Khyati Patel is a graduate student at the University of West Florida. She received her Bachelor's in Software Engineering from the University of West Florida in 2012. Her research interests are Software Engineering, DevOps, and DevOps Security.

Nathan Cooper is a Software Engineering major at the University of West Florida. Along with research into tools for Continuous Delivery and Integration he has been actively working on developing an educational pipeline to help teach these practices. His research interests are Software Engineering and Security.

Valeria Gamboa is a senior at the University of West Florida majoring in Computer Science with a minor in Mathematics.

Bhavyansh Mishra is a junior at the University of West Florida, with a major in Computer Engineering. He has worked with tools for Automatic Infrastructure Provisioning and Deployment. He has also worked on creating a full-fledged Continuous Delivery Code Pipeline. His research interests are Automated Provisioning and Deployment, Containerization, Machine Learning, IoT and Cybersecurity.

Keenal Shah is in her second year at University of West Florida, with a major in Software Engineering. She has worked with tools for Automatic Infrastructure Provisioning and continues to research Continuous Deployment. Her interests are in Computer and Network Security, User Interface Design, and Game Design and Implementation.